R's `.Call` interface

Martin Morgan
Fred Hutchinson Cancer Research Center

17-18 February, 2011

1 Example: `composite_linkage_disequilibrium`

As a simple illustration of the `.Call` interface, the following function produces a wrapper around the composite linkage disequilibrium calculation introduced in the discussion of `.C`. The function takes *R* objects (rather than just the C representation of the data in the object; lines 1, 2), checks that the objects are of the right type (lines 5-12), extracts relevant information (e.g., the dimensions of the matrix of SNPs; lines 15-17), allocates memory for the result (lines 20-21), calls the original C function (lines 24-27), and arranges to clean up and return the result (lines 29, 30).

```
1  SEXP
2  composite_linkage_disequilibrium_1(SEXP snp_r, SEXP width_r)
3  {
4      /* check incoming types */
5      if (!IS_INTEGER(width_r) ||
6          LENGTH(width_r) != 1 ||
7          INTEGER(width_r)[0] == NA_INTEGER)
8      {
9          error("'width' must be a single non-NA integer");
10     }
11     if (!IS_RAW(snp_r) || LENGTH(GET_DIM(snp_r)) != 2)
12         error("'snp' must be matrix of raw()");
13
14     /* retrieve inputs */
15     int width = INTEGER(width_r)[0],
16         n_sub = INTEGER(GET_DIM(snp_r))[0],
17         n_snp = INTEGER(GET_DIM(snp_r))[1];
18
19     /* allocate memory for return */
20     SEXP delta_r;
21     PROTECT(delta_r = allocMatrix(REALSXP, width, n_snp - width));
22
23     /* do the calculation */
24     composite_linkage_disequilibrium(RAW(snp_r),
25                                      &n_sub, &n_snp, &width,
```

```
26                                        REAL(delta_r));
27
28      /* unprotect and return */
29      UNPROTECT(1);
30      return delta_r;
31  }
```

2 Key components

S-Expressions *R* objects like `numeric()` or `integer()` are actually represented at the C level as a data structure called an *S-expression*, or `SEXP`. These are defined in the file `R_HOME/include/Rinternals.h`. They consist of book-keeping information (e.g., what type of data the `SEXP` holds, and whether there are 0, 1 or more symbols in the R session that have referred to the `SEXP`) as well as the actual data (e..g, vector of `double` or `int` values) associated with the `SEXP`.

lines 1-2 of the source code define a typical `.Call` entry point: the function takes two `SEXP` representing *R* objects, and returns a `SEXP` representing an *R* object.

Interface to *R* Internals The `SEXP` are more like *R* objects than the pointers we've seen in the `.C` entry point, and in particular they can be queried for properties such as their type (e.g., `IS_INTEGER`) or length (`LENGTH`, as in lines 5 and 6. *R* actually defines several different sets of functions for accessing objects at the C level. These interfaces are defined in `R_HOME/include/Rdefines.h` and `R_HOME/include/Rdefines.h`. Looking in these files, you'll see that the length of an object can be determined in a number of ways, including `LENGTH`, `GET_LENGTH`, `length`, and `Rf_length`. For technical reasons involving name resolution it is probably best to use the `Rf_*` interface, but in practice packages adopt different approaches.

It is informative to follow functions in either interface to their implementation, typically by checking out the *R* source code. This can be a lot of fun! For instance, `Rf_length` is defined as a function at `Rinternals.h:1047`, and implemented as an inline function at `Rinlinedfuns.h:86`, where for many types of `SEXP` it calls the *macro* `LENGTH` defined at `Rinternals.h:267`. Conversely, the *function* `LENGTH` at `Rinternals.h:379` is what *R* package code sees; it too is redefined to point to the inline implementation at `Rinlinedfuns.h:86`. And finally, in an *R* session the definition of `length` is

```
> length
function (x)  .Primitive("length")
```

R maps this to a C level function called `do_length` in a table in `src/main/names.c:196`. This function is defined in `src/main/array.c:393`, and eventually arrives at our old friend the inlined function at `Rinlinedfuns.h:86`.

Accessing data contained in *R* objects Several functions provide access to the data contained in *R* objects. We see this in lines 7 and 15-17, where the `INTEGER` function is used to retrieve a C `int*` pointer. Note that *R* uses 1-based indexing, but in C the same element is now at position 0. The C level interface provides some convenience functions, such as `GET_DIM` for retrieving the dimensions of an object that is a matrix. This actually returns a `SEXP`, which can be queried (as on line 11) for its length.

Allocating memory In *R*, one can create an object (e.g., `numeric(10)`) and not worry about what happens to the memory that is used to represent that object. One can ask *R* to allocate memory using a function like `allocMatrix` on line 21. This returns an `SEXP` containing enough room for the requested data, in this case an array of doubles.

R is managing memory, and in particular the `allocMatrix` function has asked *R* to move a portion of memory from its pool of available memory to its pool of memory that is in use. *R* periodically runs a *garbage collector* to see if any memory that is currently in use can be reclaimed for the pool of memory that can be allocated again. *R* does this by determining whether symbols in an *R* session exist that point to each block of memory. For instance, `x <- integer(10)` associates a portion of *R*'s memory with the variable `x`. When the garbage collector runs, it notes that there is a symbol `x` and that it references memory that cannot be garbage collected. If the user says `rm(x)`, then the next time the garbage collector runs it will note that the memory once pointed to by `x` is not referenced by any current symbol, and will mark the memory as suitable for garbage collection.

Line 21 allocates memory and assigns it to a C variable. But the *R* garbage collector does not know about C variables, so if the garbage collector were to run, it would think the memory could be collected. This would be very bad – the memory we thought we had access to could instead be assigned and manipulated by some other C level variable. We tell *R* not to garbage collect memory that has been allocated at the C level but has not been assigned to an *R* symbol with the `PROTECT` command, as on line 21. This puts the allocated `SEXP` on a *protection stack* that the garbage collector consults before moving memory from its in-use to its free memory pool.

Notice in line 29 that we call `UNPROTECT`. This removes the most recently added `SEXP` from the protection stack, making it available for garbage collection. Immediately after this, and before the garbage collector has a chance to run, we return the allocated memory from our `.Call` function to the *R* level. If it is assigned to an *R*-level variable, then it is again safe from garbage collection. If it is not assigned to a variable, it will be garbage collected (so we will not leak memory).

R ensures that the protection stack is in the same state when it returns from a `.Call` as when it entered it, so functions like `allocMatrix` do not result in 'memory leaks'; the more common problem is that the user forgets to `PROTECT` some allocated memory, the code functions properly most of the time, but once in a while the garbage collector runs mid-way through a `.Call` and surprising things happen. These issues can be difficult to track down.

R has additional ways in which memory can be requested. This is typically used when one wants to represent native C data types rather than *R* objects. This memory is allocated with `R_alloc` or `Calloc`, defined in `R_HOME/include/R_ext/Memory.h` and `RS.h`. Memory allocated with `R_alloc` is automatically released when *R* returns from `.Call`. In contrast, memory allocated with `Calloc` is not explicitly managed by *R*, we are responsible for releasing the memory when we are done with it, using the `Free` function. This is useful when we want to allocate memory at the C level, and use it across several calls from *R*. For instance, the function that opened a SQLite data base connection allocated memory to represent the connection, and managed this across several transitions between the *R* and C levels.

3 Directions and Resources

The "Writing R Extensions" manual has several sections on the `.Call` interface, including calling back in to *R* either for mathematical functions or to evaluate arbitrary *R* code. Chamber's *Software for Data Analysis* and Gentleman's *R Programming for Bioinformatics* include sections on calls to native languages.